A Larch Specification of Copying Garbage Collection

Scott Nettles

December 1992
CMU-CS-92-219

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Garbage collection (GC) is an important part of many language implementations. One of the most important garbage collection techniques is copying GC. This paper consists of an informal but abstract description of copying collection, a formal specification of copying collection written in the Larch Shared Language and the Larch/C Interface Language, a simple implementation of a copying collector written in C, an informal proof that the implementation satisfies the specification, and a discussion of how the specification applies to other types of copying GC such as generational copying collectors. Limited familiarity with copying GC or Larch is needed to read the specification.

This research was sponsored by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Keywords: formal methods, formal specification, garbage collection, algebraic specification, copying garbage collection, Larch specification languages, Larch/C specification language

1. Introduction

Automatic storage reclamation, or garbage collection is an important service provided by many language implementations [11]. There are two major techniques used for garbage collection, reference counting and tracing. Reference counting requires explicitly accounting for the number of references to each data item. Tracing collectors trace the pointer graph to find the reachable data. The two major variants of the tracing approach are Mark and Sweep collectors (MSGC) [7], and Copying collectors (CGC) [3]. Mark and Sweep collectors mark the data reachable from the roots as they trace out the pointer graph. They then "sweep up" the unmarked data into a free list for reallocation. Copying collection copies the reachable data of the graph to an unused portion of memory, leaving the garbage behind.

This paper presents a Larch specification and a simple implementation of copying collection as well as an informal proof that the implementation satisfies the specification. The specification itself is composed of two parts, one in the Larch Shared Language (LSL) which is used to specify general properties of CGC, and one in the Larch/C Language (LCL) which uses the LSL traits to specify all of the C routines used in the implementation. The specification should be readable even by those not familiar with Larch. For a general introduction to LSL see [6] [5] and to LCL [4]. Both the LSL and LCL specifications have been syntax and type checked, although no effort has been made at formal verification.

I do not further consider Mark and Sweep collection in detail. However since it is also based on tracing the pointer graph, those portions of the specification that deal with tracing apply to it as well. Reference counting is not addressed at all.

The paper begins with a very abstract description of how a tracing collector works, followed by a description of the standard implementation techniques used for copying collection. Next, I present the LSL and LCL specifications, followed by the implementation along with informal proofs that the implementation satisfies the specification. Finally, I discuss the applicability of the specification to several important variants of CGC, and some related work on formalizing GC.

2. Tracing and Copying Collection

The pointers contained in data form a directed graph, where the data are the nodes and the pointers are the edges. Any portion of this graph that a program cannot reach by dereferencing pointers is inaccessible to the program. Such inaccessible data is called *garbage* and can be reallocated, while any data that is accessible is called *live* and must be preserved. Tracing collectors find the live data by computing the transitive closure of the *points-to* relation starting from the set of known live data, called *roots*. The differences among tracing collectors lie in what algorithm is used to compute the transitive closure, and what is done to the live data when they are found by the algorithm. Algorithms for computing transitive closures are graph searching algorithms, and not surprisingly MSGC uses a depth-first search, and CGC a breadth-first search. (But see Section 5 for some exceptions to this rule.) Both may use clever representation techniques to avoid using extra storage beyond that needed for the data while computing the transitive closure.

The following is a general graph searching algorithm. Nodes are divided into two disjoint sets: the *seen* nodes, which are known to be in the transitive closure, and the *unseen* nodes, which may or may not be. The seen nodes are further divided into two disjoint sets: the *visited* nodes, which have had the nodes they refer to added to the seen set, and the *unvisited* nodes, which have not. The algorithm starts by placing all the roots in the unvisited set: all other nodes are in the unseen set. It proceeds by selecting some member of the unvisited set, adding the nodes that it refers to that are unseen to the unvisited set, and then adding the node to the visited set. When the unvisited set is empty, the algorithm terminates, and all of the live nodes are in the visited set. Depth-first search of the graph results from managing the unvisited set as a stack and breadth-first search results from managing it as a queue.

In addition to performing some variant of the algorithm above, tracing collectors perform some additional actions when a node is added to the seen set. For MSGC this consists of marking the node so that the reachable nodes can be distinguished from the unreachable ones during the sweep phase. For CGC this consists of copying the node to a new location in memory. Since other nodes may still refer to the original node, when a node is copied the original node must be modified so that the fact that it has been copied can be detected, and where it has been copied to can be found. This is usually done by marking the node as "forwarded" using a tag and writing a forwarding pointer into the data indicating where it was copied to.

The usefulness of CGC comes in part from the use of a clever encoding of the unseen, unvisited and visited sets so that no more memory is used by the algorithm than is needed to copy just the live data. The unseen and seen sets are encoded by placing them in different portions of memory. *From-space* holds the unseen set and is where data is copied from; *to-space* holds the seen set and is where the data is copied to. Typically CGC visits the data in the graph in a breadth-first manner, and thus the unvisited set must form a queue. To effect a queue, CGC uses two pointers into to-space, the *unscanned* pointer and the *scanned* pointer. The unscanned pointer points to the first location of to-space that is unused and it forms the tail of the queue. Data is added to the seen set by copying it to the location referred to by the unscanned pointer. The scanned pointer points to the location of the first unvisited node, and forms the head of the queue. Because of the use of unscanned and scanned pointers, CGC terminology generally uses the term unscanned for unvisited, and scanned for visited.

The standard CGC algorithm is known as the Cheney scan [1]. It utilizes three basic operations: *copying*, *forwarding*, and *scanning*. *Copying* copies a node to the location referred to by the unscanned pointer and sets the unscanned pointer to refer to the first location after the newly copied data. It also modifies the original data to record the fact that the data has been copied, as well as the location it was copied to. This is exactly the act of adding the node to the seen set. *Forwarding* modifies a pointer to from-space data so that it refers to the to-space copy of the data. If the node has not yet been copied, it copies it. *Scanning* a node forwards each pointer in the node and advances the scanned pointer so that it refers to the next node in to-space. Since *forwarding* guarantees that a node has been copied, *scanning* corresponds directly to adding the node to the visited set. In addition *scanning* guarantees that no pointers into from-space are found in scanned nodes.

Given the operations and data structures above, the actual garbage collection algorithm is very simple. When the user program (known as the mutator) runs out of storage, the garbage collector is called. The roots are defined in an implementation dependent manner, and the unscanned and scanned pointers are directed at the beginning of to-space. Next, each root is forwarded. This causes all directly reachable nodes to be copied into the unscanned (seen and unvisited) set and updates the roots so that they point to the new copies. It does not change the scanned pointer. Now the node pointed to by the scanned pointer is scanned and the scanned pointer is advanced past the newly scanned node. This is repeated until the scanned pointer equals the unscanned pointer, which indicates the queue is empty. When this happens the roles of the two spaces are exchanged ("flipped") and the mutator can resume. This process examines each live node twice, once to copy it and once to scan it, and thus the cost of the algorithm is proportional to the number of live nodes. The live nodes are copied into a contiguous region of memory, which serves to compact memory.

3. The Specification

All of the key concepts and terminology needed to understand the specification have been introduced. The specification itself is made up of two kinds of components, LSL traits and LCL interfaces. The LSL traits define sorts and functions at a high level of abstraction and form the vocabulary used in the interfaces. The LCL interfaces specify pre-conditions that must be satisfied before the routine may be used, and post-conditions that the routine must guarantee upon termination.

I first present the traits containing the key sorts and some important general functions. Then I present the LCL interfaces in a top-down fashion along with the supporting LSL traits. The Appendix contains several of the less important traits which I do not discuss here.

Many of the LSL functions take the form op(arg, arg'), which specifies a relation between a pre-state and a post-state. In LCL interfaces pre-states are notated with a^ and post-states with a '.

3.1. Address trait

Address : **trait**
  includes $Set(A, SA)$ % Sets of Addresses

Figure 1: The Address Trait

Addresses (A) are used to "index" memory. They can only be compared for equality, since no other operations are defined on them.

3.2. Node trait

Node : **trait**
  includes *Address*
  includes $Set(Val, SV)$ % Sets of Values
  includes $Set(N, SN)$ % Sets of Nodes
  N **tuple of** $id : UID, addrs : SA, vals : SV$

Figure 2: The Node Trait

Nodes (N) are the basic data items of the specification. They consist of a unique identifier, a set of addresses that are the addresses of the other nodes "pointed to" by the node, and a set of values representing the non-pointer data in a node.

3.3. Memory trait

A memory (M) (figure 3) consists of four sets of addresses and two maps. *Roots* is the set of root addresses. *Uncopied*, *unscanned*, *scanned* are the sets of addresses that are uncopied, unscanned, and scanned. Collectively unscanned and scanned are the addresses that have been copied. The *mem* map maps addresses to nodes, while the *forwarded* map maps the original address of a copied node to its new address.

isValidMemory captures the notion that a memory is well-formed. It is an invariant of all the LCL interfaces. Since it is the first function we have seen, and an important one as well, let's examine it in detail. The line

$$isOneToOne(m.mem) \wedge isOneToOne(m.forwarded)$$

says that only one node can be located at any given address in memory, and that only one node can be forwarded to any given address. The line

$$isValidAddrSet(m.roots, m)$$

says that all the roots are addresses located in memory. The lines

$$m.uncopied \cap m.unscanned = \{\}$$
$$m.uncopied \cap m.scanned = \{\}$$
$$m.unscanned \cap m.scanned = \{\}$$

say that the uncopied, unscanned, and scanned sets are all disjoint. The line

$$m.uncopied \cap domain(m.forwarded) = \{\}$$

says that no uncopied address has been forwarded. The line

$$m.unscanned \cup m.scanned = range(m.forwarded)$$

says that the addresses which have been copied are exactly those which are mapped to by the forwarded map.

MemoryMain : **trait**

includes *Node*

includes *FiniteMappingAux*(*ANMap*, *A*, *N*, *SA* **for** *SDomain*)

includes *FiniteMappingAux*(*AAMap*, *A*, *A*, *SA* **for** *SRange*, *SA* **for** *SDomain*)

includes *TestSet1Arg*(*isValidAddr*, *isValidAddrSet*, *A*, *SA*, *M*)

M **tuple of** *roots* : *SA*,
uncopied : *SA*,
unscanned : *SA*,
scanned : *SA*,
mem : *ANMap*,
forwarded : *AAMap*

introduces

$isValidMemory : M \rightarrow Bool$

$isValidAddr : A, M \rightarrow Bool$

$effectiveAddr : A, M \rightarrow A$

asserts

$\forall\, m : M, a : A, n : N$

$isValidMemory(m) ==$
$isOneToOne(m.mem) \wedge isOneToOne(m.forwarded)$
$\wedge isValidAddrSet(m.roots, m)$
$\wedge m.uncopied \cap m.unscanned = \{\}$
$\wedge m.uncopied \cap m.scanned = \{\}$
$\wedge m.unscanned \cap m.scanned = \{\}$
$\wedge m.uncopied \cap domain(m.forwarded) = \{\}$
$\wedge m.unscanned \cup m.scanned = range(m.forwarded)$
$\wedge m.uncopied \cup m.unscanned \cup m.scanned$
$= domain(m.mem)$

$isValidAddr(a, m) ==$ **if** $defined(m.forwarded, a)$
then $defined(m.mem, m.forwarded[a])$
else $defined(m.mem, a)$

$effectiveAddr(a, m) ==$ **if** $defined(m.forwarded, a)$
then $m.forwarded[a]$
else a

implies

converts *isValidMemory*, *isValidAddr*, *isValidAddrSet*, *effectiveAddr*

Figure 3: The MemoryMain Trait

Finally $m.uncopied \cup m.unscanned \cup m.scanned = domain(m.mem)$ says that all nodes are referred to by an address in the uncopied, unscanned, or scanned sets.

effectiveAddr translates unforwarded address to forwarded ones, if the node has been copied. The MemoryAuxiliary trait, found in the appendix, defines many simple functions involving memory, mostly serving to improve the specification's readability.

3.4. Equiv trait

Equiv : **trait**
  includes *Memory*
  includes $PairwiseElementTest2Arg(isEquivAddr, A, A, SA, SA, M, M,$
    $addrsEquiv$ **for** $allPass)$
  introduces
    $isEquivAddr : A, A, M, M \rightarrow Bool$
    $isEquivNode : N, N, M, M \rightarrow Bool$
    $memEquiv : M, M \rightarrow Bool$
  asserts
    $\forall\ m, m' : M, a, a' : A, n, n' : N$
      $isEquivAddr(a, a', m, m') ==$
        $effectiveAddr(a, m') = effectiveAddr(a', m')$
        $\wedge isEquivNode(nodeAtAddr(a, m), nodeAtAddr(a', m'), m, m')$

      $isEquivNode(n, n', m, m') ==$
        $n.id = n'.id$
        $\wedge n.vals = n'.vals$
        $\wedge addrsEquiv(n.addrs, n'.addrs, m, m')$

      $memEquiv(m, m') ==$
        $isValidMemory(m) \wedge isValidMemory(m')$
        $\wedge addrsEquiv(m.roots, m'.roots, m, m')$
        $\wedge addrsEquiv(allNodes(m), allNodes(m'), m, m')$
  implies
    converts $isEquivAddr, isEquivNode, addrsEquiv, memEquiv$

Figure 4: The Equiv Trait

The *Equiv* trait captures the notion of equivalence between two addresses, two nodes or two memories. Two addresses are equivalent if they are equal or one is the forwarded version of the other and the nodes they point to are equivalent. Two nodes are equivalent if they have the same UID and values and if the addresses contained in them are equivalent. Two memories are equivalent if they are both well formed and their roots and all the nodes are equivalent. In addition to the functions directly defined, the function *addrsEquiv* is defined by including the trait *PairwiseElementTest2Arg* with the function *isEquivAddr*, which is used to test that all elements of one set have equivalent addresses in another.

3.5. Reachable trait

$$
\begin{array}{l}
\textit{Reachable} : \textbf{trait} \\
\quad \textbf{includes } \textit{Memory} \\
\quad \textbf{introduces} \\
\quad\quad \textit{reachable} : SA, M \rightarrow SA \\
\quad\quad r_1 : SA, SA, M \rightarrow SA \\
\quad \textbf{asserts} \\
\quad\quad \forall\ m : M, a : A, as, as_1, as_2 : SA \\
\quad\quad\quad \textit{reachable}(as, m) == r_1(\{\}, as, m) \\
\\
\quad\quad\quad r_1(as, \{\}, m) == as \\
\quad\quad\quad r_1(as_1, \textit{insert}(a, as_2), m) == \\
\quad\quad\quad\quad r_1(\textit{insert}(a, as_1), \\
\quad\quad\quad\quad (as_2 \cup (m.mem[a]).addrs) - \textit{insert}(a, as_1), m) \\
\quad \textbf{implies} \\
\quad\quad \textbf{converts } \textit{reachable}, r_1
\end{array}
$$

Figure 5: The Reachable Trait

The Reachable trait is the heart of the specification: all data reachable from the roots is live. Reachability is the transitive closure of the "points to" relation starting from some given set of addresses. *reachable* is defined using the helper function *r1*. The first two arguments to *r1* are the visited and unvisited sets respectively. *reachable* invokes *r1* with the initial addresses in the unvisited set. The main action of *r1* is to transfer nodes from the unvisited to the visited set. When a node is transferred to the visited set, all the addresses directly referred to by it are added to the unvisited set minus any addresses already in the visited set. When the unvisited set is empty, *r1* is done. No order of addition to either set is implied, and thus *r1* does not specify any fixed search order.

3.6. GC

This section begins the presentation of the main body of the specification. The specification is presented in a top-down fashion. Both the LCL interfaces and LSL traits are discussed. I typically present several LCL interfaces that share a common trait, followed by the trait itself. This allows the reader to see how a function is used before seeing the details of the function itself. A functions name should give some insight into its semantics.

```
imports base;
uses GC(memory for M, addr for A );

void gc(void) memory mem; {
    requires isInitialMemory(mem^);
    modifies mem;
    ensures
        isFullGC(mem^, mem')
      /\ isFinalGCMemory(mem');
}
```

Figure 6: gc Interface

gc is the primary interface to the garbage collector. It performs a garbage collection but stops before the spaces are "flipped". The pre-condition is that the memory be in its pre-gc state, i.e., essentially that nothing is yet copied. The post-condition is that all the reachable data have been copied and the memory is in its post-gc state.

```
imports base;
uses GC(memory for M, addr for A);

void finalizeGC(void) memory mem;  {
  requires isFinalGCMemory(mem^);
  modifies mem;
  ensures
       isInitialMemory(mem')
    /\ mem^.scanned = mem'.uncopied
    /\ mem^.roots = mem'.roots
    /\ mem^.mem = mem'.mem;
}
```

Figure 7: finalizeGC Interface

finalizeGC "flips" the spaces. The pre-condition is that a GC has just completed, and the post-condition requires that the implementation ensure that the memory is in a state where the mutator can resume.

GC : **trait**
  includes *Memory*
  includes *Equiv*
  includes *Reachable*
  introduces
    $isFullGC : M, M \rightarrow Bool$
    $isInitialMemory : M \rightarrow Bool$
    $isFinalGCMemory : M \rightarrow Bool$
  asserts
    $\forall\ m, m' : M$
      $isFullGC(m, m') ==$
        $isInitialMemory(m)$
        $\wedge isFinalGCMemory(m')$
        $\wedge memEquiv(m, m')$
        $\wedge addrsEquiv(reachable(m.roots, m), m'.scanned, m, m')$

      $isInitialMemory(m) ==$
        $isValidMemory(m)$
        $\wedge \{\} = m.unscanned$
        $\wedge \{\} = m.scanned$
        $\wedge \{\} = m.forwarded$

      $isFinalGCMemory(m) ==$
        $isValidMemory(m)$
        $\wedge \{\} = m.unscanned$
        $\wedge \{\} = rootsUnforwarded(m)$
  implies
    converts *isFullGC*, *isInitialMemory*, *isFinalGCMemory*

Figure 8: GC Trait

The *GC* trait captures the essential requirements of a copying collector. Initially the memory must be entirely uncopied. When a GC completes, all the reachable data must have been copied to the scanned set,

the roots updated, the unscanned set empty, and otherwise the memories are still equivalent. All unreachable nodes are left in the uncopied set.

3.7. Roots

```
imports base;
uses GC(memory for M, addr for A);

void forwardRoots(void) memory mem;  {
  requires isInitialMemory(mem^);
  modifies mem;
  ensures
       {} = rootsUnforwarded(mem')
    /\ mem'.roots = mem'.unscanned
    /\ {} = mem'.scanned
    /\ memEquiv(mem^, mem');
}
```

Figure 9: forwardRoots Interface

forwardRoots is responsible for forwarding the roots. The pre-condition is that the memory has not yet had anything copied. The post-condition is that all of the roots have been forwarded and are in the unscanned set but that otherwise memory is unchanged.

```
imports base;

addr nextUnforwardedRoot(void) memory mem; {
  requires isValidMemory(mem^);
  ensures if {} = rootsUnforwarded(mem^)
          then result = aNIL
          else result \in rootsUnforwarded(mem^);
}
```

Figure 10: nextUnforwardedRoot interface

nextUnforwardedRoot returns an unforwarded root if one exists, aNil otherwise. *aNil* is just a user defined LCL constant for a nil address.

```
uses Forward(memory for M, addr for A );

void forwardRootAddr(addr *a) memory mem; {
  requires
       isValidMemory(mem^) /\ (*a)^ \in rootsUnforwarded(mem^);
  modifies mem, *a;
  ensures
       mem'.roots = (mem^.roots - {(*a)^}) \U {(*a)'}
    /\ isForwardStep((*a)^, (*a)', mem^, mem');
}
```

Figure 11: The forwardRootAddr interface

forwardRootAddr forwards a single root. The pre-condition is that the address be an unforwarded root. The post-condition is that the address is forwarded, and that its new value replaces the old value in the roots. *isForwardStep* is defined in the *Forward* trait found below in figure 17.

3.8. Scanning

```
imports base;

void scanUnscanned(void) memory mem;  {
  requires
        {} = rootsUnforwarded(mem^)
    /\ mem^.roots = mem^.unscanned
    /\ {} = mem^.scanned
    /\ isValidMemory(mem^);
  modifies mem;
  ensures
        mem^.roots = mem'.roots
    /\ mem'.unscanned = {}
    /\ memEquiv(mem^, mem')
    /\ addrsEquiv(reachable(mem^.roots, mem^),
                  mem'.scanned, mem^, mem');
}
```

Figure 12: The scanUnscanned Interface

scanUnscanned completes the transitive closure calculation starting from the forwarded roots. It requires that the roots all be forwarded and that nothing is scanned. The post-condition is that scanning is complete, and that all nodes reachable from the initial roots have been copied and scanned, but that otherwise the memories are equivalent.

```
imports base;

addr nextUnscannedNode(void) memory mem;  {
  requires isValidMemory(mem^);
  ensures if {} = mem^.unscanned
          then result = aNIL
          else result \in mem^.unscanned;
}
```

Figure 13: The nextUnscannedNode interface

nextUnscannedNode must return an unscanned node unless there are none left, in which case it must return *aNIL*.

scanAddr (figure 14) scans a single address. The pre-condition is that the address be unscanned and that the nodes reachable from the roots also be reachable from the copied set. The post-condition is that the address has been scanned, and that the nodes reachable from the roots are still reachable from the copied set. New nodes may have been added to the copied set.

```
imports base;
uses Scan(memory for M, addr for A);

void scanAddr(addr a) memory mem;  {
  requires
        isValidMemory(mem^)
    /\ addrUnscanned(a, mem^)
    /\ addrsEquiv(reachable(mem^.roots, mem^),
                  reachable(copiedNodes(mem^), mem^),
                  mem^, mem^);
  modifies mem;
  ensures
        isScanStep(a, mem^, mem')
    /\ addrsEquiv(reachable(mem^.roots, mem^),
                  reachable(copiedNodes(mem'), mem'),
                  mem^, mem');
}
```

Figure 14: The scanAddr interface

Scan : **trait**
    includes *Memory*
    includes *Forward*
    introduces
        $isScannedAddr : A, M \rightarrow Bool$
        $isScanStep : A, M, M \rightarrow Bool$
    asserts
        $\forall\ m, m' : M, a : A$
            $isScannedAddr(a, m) ==$
                $addrScanned(a, m)$
                $\wedge isForwardedAddr(a, a, m, m)$
                $\wedge isForwardedSet((m.mem[a]).addrs,$
                $(m.mem[a]).addrs, m, m)$

            $isScanStep(a, m, m') ==$
                $memEquiv(m, m')$
                $\wedge m.roots = m'.roots$
                $\wedge addrUnscanned(a, m)$
                $\wedge addrScanned(a, m')$
                $\wedge isForwardedSet((m.mem[a]).addrs,$
                $(m'.mem[a]).addrs, m, m')$
    implies
        $\forall\ m, m' : M, a : A$
            $isScanStep(a, m, m') \Rightarrow isScannedAddr(a, m')$

        converts *isScannedAddr, isScanStep*

Figure 15: The Scan trait

The *Scan* trait defines functions used to describe scanning. The function *isScannedAddr* is true if the address is in the scanned set, and if ι oth it and its references have been forwarded. The function *isScanStep* relates two memories that differ only in that one step of scanning has occurred. This means that the forwarded address of the node is added to the scanned set, and that all of its referents are scanned. *isScanStep* iп. ..*ScannedAddr*.

3.9. Forwarding

```
imports base;
uses Forward(memory for M, addr for A);

void forwardAddr(addr *a)  memory mem; {
    requires isValidMemory(mem^);
    modifies mem, *a;
    ensures
        if isForwardedAddr((*a)^, (*a)', mem^, mem')
        then (*a)^ = (*a)' /\ mem^ = mem'
        else isForwardStep((*a)^, (*a)', mem^, mem');
}
```

Figure 16: The forwardAddr interface

forwardAddr forwards an address if it has not already been forwarded. If it has been forwarded, then nothing changes. The post-condition is that an unforwarded address is forwarded. Allowing *forwardAddr* to be applied to already forwarded addresses gives additional flexibility to the specification which will be discussed later.

Forward : trait
    includes *Memory*
    includes *Copy*
    includes $PairwiseElementTest2Arg(isForwardedAddr, A, A, SA, SA, M, M,$
        $isForwardedSet$ for $allPass)$
    introduces
        $isForwardedAddr : A, A, M, M \rightarrow Bool$
        $isForwardStep : A, A, M, M \rightarrow Bool$
    asserts
        $\forall\ m, m' : M, a, a' : A$
            $isForwardedAddr(a, a', m, m') ==$
                $isCopiedAddr(a, a', m, m')$
                $\wedge effectiveAddr(a, m') = a'$

            $isForwardStep(a, a', m, m') ==$
                $memEquiv(m, m')$
                $\wedge addrUnforwarded(a, m)$
                $\wedge addrForwarded(a', m')$
                $\wedge (addrUncopied(a, m) \Rightarrow isCopyStep(a, m, m'))$
                $\wedge isCopiedAddr(a, a', m, m')$
                $\wedge m'.forwarded[a] = a'$
                $\wedge m.scanned = m'.scanned$
    implies
        $\forall\ m, m' : M, a, a' : A$
            $isForwardStep(a, a', m, m') \Rightarrow isForwardedAddr(a, a', m, m')$

        converts $isForwardedAddr, isForwardStep$

Figure 17: The Forward trait

The *Forward* trait defines functions used to describe forwarding. *isForwardedAddr* says that the address must be copied, and that at least the post version of the address (a') must refer to the copied version of the node. The indirectly defined *isForwardedSet* function says that for every address in one set of addresses, some address in the other set satisfies *isForwardedAddr*. *isForwardStep* relates an unforwarded address and a memory to a forwarded address, and a memory in which only the changes needed to forward the address have occurred. If the address has not been copied, it is. The new address refers to the copy.

3.10. Copying

```
imports base;
uses Copy(memory for M, addr for A );

void copyAddr(addr a)  memory mem;  {
  requires
        isValidMemory(mem^)
    /\ addrUncopied(a, mem^);
  modifies mem;
  ensures isCopyStep(a, mem^, mem');
}
```

Figure 18: The copyAddr interface

copyAddr copies an uncopied address to a free location. No other changes are made to memory.

Copy : **trait**
  includes *Memory*
  includes *Equiv*
  introduces
    $isCopiedAddr : A, A, M, M \rightarrow Bool$
    $isCopyStep : A, M, M \rightarrow Bool$
  asserts
    $\forall\ m, m' : M, a, a' : A$
      $isCopiedAddr(a, a', m, m') ==$
        $isEquivAddr(a, a', m, m')$
        $\wedge addrCopied(effectiveAddr(a', m'), m')$

      $isCopyStep(a, m, m') ==$
        $memEquiv(m, m')$
        $\wedge addrUncopied(a, m)$
        $\wedge addrFree((m'.forwarded[a]), m)$
        $\wedge addrUnforwarded(m'.forwarded[a], m)$
        $\wedge m' = [m.roots,$
        $delete(a, m.uncopied),$
        $insert(m'.forwarded[a], m.unscanned),$
        $m.scanned,$
        $rebind(m.mem, a, m'.forwarded[a]),$
        $bind(m.forwarded, a, m'.forwarded[a])]$
  implies
    $\forall\ m, m' : M, a : A$
      $isCopyStep(a, m, m') \Rightarrow isCopiedAddr(a, a, m, m')$

    converts *isCopiedAddr*, *isCopyStep*

Figure 19: The Copy trait

The *Copy* trait defines *isCopiedAddr* and *isCopyStep*. *isCopiedAddr* is true if *a* and *a'* are equivalent, and the node they refer to has been copied in *m'*. *isCopyStep* says *a* is uncopied and the address it is to be copied to is free and unforwarded. The memory after copying (*m'*) is related to memory before copying (*m*) in the following way: the roots and scanned sets are unchanged, *a* is removed from the uncopied set, and its new location added to the uncopied set, the node referred to by *a* is now found at the new address, and the forwarded map has a bound to its new location. *isCopyStep* only constrains the new location of the node to be free but says nothing about nodes being copied to contiguous addresses.

4. Implementation and Informal Proof of Correctness

The implementation is simple, designed to be short and easy to understand without sacrificing any details fundamental to the algorithm. All nodes are "cons" cells containing no data fields and two pointer fields, car and cdr. Space is allocated for the forwarding pointer explicitly rather than using some part of the node data as probably would be done in a real collector. Data representation issues such as tagging pointers, node lengths, etc., while important in a real language implementation, are not essential to capturing the essence of the copying collection algorithm and are thus ignored.

Originally I had not planned on proving the implementation correct, even in the informal manner done here. However as the specification proceeded, I found it very difficult to convince myself that I had both included and excluded the right things. Informally verifying the implementation caused me to make significant modifications to both the LSL and LCL portions of the specification, and gave me greatly increased confidence that the specifications are essentially correct. At this point only a complete formal verification would increase my confidence significantly, and even then I would be surprised if it induced more than minor modifications.

The presentation follows the same top-down order as that of the specification. First, I present the implementation's representation memory in the form of the include file gc.h. This is followed by a discussion of the abstraction function which maps between the representation of memory used in the implementation and that used in the specification, as well as an invariant which must be preserved by the implementation. This invariant is needed for some of the proofs. I then present the implementation of each of the interfaces, along with the informal proof that it satisfies its specification. Unfortunately, this portion of the paper is difficult to read as it requires frequent back references to the specifications. The complete specification, found in the Appendix, may be easier to refer to than the specifications in the previous section. The driver code used to test the garbage collector is omitted.

4.1. The Representation of Memory

The include file gc.h captures the implementation's representation of memory and plays the same role as the Address, Node, and Memory traits (figures 1, 2, 3).

```
#define maxNumRoots 4
#define maxNumNodes 12

typedef int addr;

typedef enum {CONS, FWD} tag_t;

typedef struct {
  tag_t tag;
  addr fwd;
  addr car;
  addr cdr;
} node;
```

```
typedef struct {
  addr roots[maxNumRoots];
  node to[maxNumNodes];
  node from[maxNumNodes];
  addr unscanned;
  addr scanned;
  addr alloc;
  addr next_root;
} memory;

extern const addr aNIL;
```

Addresses are simply indices into arrays. *Nodes* are structs with fields for a tag, a forwarding address, a car address, and a cdr address. If the tag is CONS then the node is uncopied and the car and cdr field hold valid pointers. If the tag is FWD then the node has been copied and the fwd field holds the to-space address of the copy. The *memory* struct closely mirrors the *Memory* trait. The root array holds the roots; only elements which are not aNil are actually roots. The to and from arrays form to-space and from-space and together make up the mem and forward maps. Any node in from-space which has a tag FWD is part of the fwd map, while all other from-space nodes and all to-space nodes are part of the mem map. To-space is divided into the scanned and unscanned sets by the scanned pointer, while the next free location in to-space is indicated by the unscanned pointer. Next_root is used during the forwarding of the roots to keep track of the next root to forward. Alloc indicates the next free location during mutation, and bounds the valid nodes in from-space.

Addresses are just integers. Thus it is impossible to tell if an address should be used as an index into the from array or the to array just by examining it. Because of this ambiguity the implementation must be careful to keep track of which array an address refers to. This gives rise to an important set of invariants which the implementation must maintain. For the root array the addresses located at indices in the range [0..next_root) refer to the to array, while those at indices in the range [next_root..max_roots) refer to the from array. (The notation [m..n) denotes the set of addresses including m, but excluding n.) In the from array, nodes with tag CONS contain references into the from array in their car and cdr fields, and nodes with tag FWD contain references into the to array in their fwd field. In the to array, all nodes at addresses in the range [0..scanned) are forwarded and contain only references into the to array, while all nodes at addresses in the range [scanned..uncopied) are unforwarded and contain only references into the from array. These conditions are invariants and each routine in the implementation may assume they hold at the beginning of its execution and must guarantee that they hold at the end. The proofs will argue that these conditions are maintained.

Now consider the correspondence between the implementation and the specification representations of addresses, nodes and memory in a somewhat more formal light. The ambiguity noted above implies that implementation addresses do not uniquely correspond to addresses in the specification. The invarients given above allow us to disambiguate. An implementation node with a tag of CONS corresponds directly to a node in the specification, with the car and cdr fields making up the address set of the specification node. The implementation representation does not contain an explicit UID and the set of values is empty. Now consider how each component of the specification's memory can be derived from the implementation's representation. M indicates the specification's representation of memory, I have used the component names of the implementations memory directly. First consider the components of M which are sets.

M.roots = {a $\in$ [0..max_roots) | roots[a] != aNil }
M.uncopied = {a $\in$ [0..alloc) | from[a].tag = CONS }
M.unscanned = [scanned..unscanned)
M.scanned = [0..scanned)

M.mem consists of the map that maps all the addresses in M.uncopied to the nodes in the from array at those addresses and all the valid addresses in the to array to the nodes in the to array.

∀ a ∈ M.uncopied . M.mem[a] = from[a]
∀ a ∈ [0..unscanned) . M.mem[a] = to[a]

Finally M.forward consists of the map which maps all the addresses in the from array which refer to forwarded nodes to the addresses in those nodes fwd field:

∀ a ∈ {b ∈ [0..alloc) | from[b].tag = FWD} . M.forward[a] = from[a].fwd

4.2. gc

This section begins the top down presentation of the code and the informal proof of correctness. The implementation itself is very simple and will not be commented on extensively. The arguments that the invariant *isValidMemory* is maintained have been omitted as they are obvious but long and tedious.

```
void gc(void)
{
    forwardRoots();
    scanUnscanned();
}
```

To show that *gc* satisfies its specification (figure 6) the following must be true: the pre-condition of *gc* implies the pre-condition of *forwardRoots*, the post-condition of *forwardRoots* implies the pre-condition of *scanUnscanned*, and the post-condition of *scanUnscanned* implies the post-condition of *gc*.

The first point is trivial, since the pre-condition of *gc* is the same as the pre-condition of *forwardRoots*. The second point follows directly from the fact the first three conjuncts of the post-condition of *forwardRoots* are the same as the first three conjuncts of the pre-condition of *scanUnscanned* and the last conjunct of the post-condition of *forwardRoots* (memEquiv(mem^, mem')) directly implies the last conjunct of the pre-condition of *scanUnscanned* (isValidMemory(mem^)). The mem' in the post-condition of *forwardRoots* is the same as mem^ in pre-condition of *scanUnscanned*.

The final point is also straightforward. Let the state of memory before any execution be m, after executing *forwardRoots* be m', and after executing *scanUnscanned* be m". After expanding *isFullGC* and *isFinalGCMemory*, adding some facts from the post-condition of *forwardRoots*, and eliminating any conjuncts which follow directly from the pre-conditions, it must be shown that:

$$\begin{aligned}
&\{\} = rootsUnforwarded(m') \wedge m'.roots = m'.unscanned \\
&\wedge \{\} = m'.scanned \wedge memEquiv(m, m') \wedge m'.roots = m''.roots \\
&\wedge m''.unscanned = \{\} \wedge memEquiv(m', m'') \\
&\wedge addrsEquiv(reachable(m'.roots, m'), m''.scanned, m', m'') \\
&\Rightarrow memEquiv(m, m'') \\
&\wedge addrsEquiv(reachable(m.roots, m), m''.scanned, m, m'') \\
&\wedge isValidMemory(m'') \wedge \{\} = m''.unscanned \\
&\wedge \{\} = rootsUnforwarded(m'')
\end{aligned}$$

From the above one can conclude that all of the following hold

$$memEquiv(m, m') \wedge memEquiv(m', m'') \Rightarrow memEquiv(m, m'')$$

$$\begin{aligned}
&memEquiv(m, m') \wedge m'.roots = m''.roots \\
&\wedge addrsEquiv(reachable(m'.roots, m'), m''.scanned, m', m'') \\
&\Rightarrow addrsEquiv(reachable(m.roots, m), m''.scanned, m, m'')
\end{aligned}$$

$$memEquiv(m', m'') \Rightarrow \{\} = m''.unscanned$$

$\{\} = rootsUnforwarded(m') \wedge m'.roots = m''.roots$
$\Rightarrow \{\} = rootsUnforwarded(m'')$

and thus that the post-condition of *scanUnscanned* implies post-condition of *gc*. Therefore *gc* satisfies its specification.

4.3. finaliseGC

```
void finalizeGC()
{
  addr i;

  for (i = 0; i < mem.scanned; i++) {
    mem.from[i] = mem.to[i];
  }

  mem.alloc = mem.scanned;
  mem.next_root = mem.scanned = mem.unscanned = 0;
}
```

finalizeGC is used to "flip" the spaces after *gc* has completed. The pre-condition for *finalizeGC* is satisfied if it follows *gc*. For *finalizeGC* to satisfy its specification (figure 7) the post-condition (isInitialMemory(mem') ∧ mem^.scanned = mem'.uncopied ∧ mem^.roots = mem'.roots ∧ mem^.mem = mem'.mem) must hold after *finalizeGC* executes.

The for loop copies scanned to uncopied without changing any addresses, satisfying mem^.scanned = mem'.uncopied and mem^.mem = mem'.mem. The roots are not changed, so mem^.roots = mem'.roots holds. *isInitialMemory* holds for the the following reasons. Setting scanned and unscanned to 0 means the scanned and unscanned sets are empty. None of the nodes which were in mem.to and which were copied into mem.from had a tag FWD, so the forwarded map is empty. In a more typical implementation the copy probably would not be done, the "flip" might be accomplished purely by changing pointers.

4.4. forwardRoots

```
void forwardRoots(void)
{
  addr r;

  while ((r = nextUnforwardedRoot()) != aNIL) {
    forwardRootAddr(&mem.roots[r]);
  }
}
```

To show that *forwardRoots* satisfies its specification (figure 9), it must be shown that assuming the pre-condition and loop termination then the post-condition is satisfied (partial correctness), and that the loop terminates. Showing partial correctness of the loop requires a loop condition (LC), and loop invariant (LI), while showing loop termination requires a metric (M) which decreases monotonically with each iteration of the loop.

$LC == \{\} != rootsUnforwarded(mem')$
$LI == memEquiv(mem\hat{}, mem') \wedge \{\} = mem'.scanned$

$\wedge$ *mem'.unscanned* $\subset$ *mem'.roots*
$M == size(rootsUnforwarded(mem'))[>= 0]$

LI is true before the loop executes assuming the pre-condition because
$mem\hat{} = mem' \Rightarrow memEquiv(mem\hat{}, mem')$
$isInitialMemory(mem) \Rightarrow \{\} = mem.scanned$
$isInitialMemory(mem) \Rightarrow \{\} = m.unscanned$
$\{\} = m.unscanned \Rightarrow mem.unscanned \subset mem.roots$

LI implies that the pre-condition for *nextUnforwardedRoot* holds, and the post-condition of *nextUnforwardedRoot* guarantees that either the pre-condition for *forwardRootAddr* holds, or that the loop terminates.

The post-condition of *forwardRootAddr* along with the fact that a is an unforwarded root implies that LI remains true because:
$\{\} = mem\hat{}.scanned \wedge mem'.roots = (mem\hat{}.roots - (*a)\hat{}) \cup (*a)'$
$\wedge\ isForwardStep((*a)\hat{}, (*a)', mem\hat{}, mem')$
$\Rightarrow memEquiv(mem\hat{}, mem') \wedge \{\} = mem'.scanned$
$\wedge\ mem'.unscanned \subset mem'.roots$

If the loop terminates then ¬LC ∧ LI holds, which satisfies the post-condition of *forwardRoots* because:
$\neg LC \wedge LI == \{\} = rootsUnforwarded(mem')$
$\wedge\ memEquiv(mem\hat{}, mem')$
$\wedge\ \{\} = mem'.scanned \wedge mem'.unscanned \subset mem'.roots$

$\{\} = rootsUnforwarded(mem') \wedge \{\} = mem'.scanned$
$\Rightarrow mem'.roots \subset mem'.unscanned$

$mem'.roots \subset mem'.unscanned \wedge mem'.unscanned \subset mem'.roots$
$\Rightarrow mem'.roots = mem'.unscanned$

The loop terminates because each time through the loop *forwardRootAddr* causes M to decrease. When it reaches 0 the loop terminates.

4.5. nextUnforwardedRoot

```
addr nextUnforwardedRoot(void){

  while ((mem.roots[mem.next_root] == aNIL) &&
 (mem.next_root < maxNumRoots)) {
    mem.next_root++;
  }

  if (mem.next_root >= maxNumRoots) return aNIL;

  return mem.next_root;
}
```

The specification for *nextUnforwardedRoot* is found in figure 10. The code loops through the roots, until it either finds an entry which is not aNil which it then returns, or it runs out of roots in which case it returns aNil. The result is an unforwarded root if one remains and aNil otherwise, thus satisfying the post-condition.

4.6. forwardRootAddr

```
void forwardRootAddr(addr *r){
  assert(r == &mem.roots[mem.next_root]);
  forwardAddr(r);
  mem.next_root++;
}
```

The specification for *forwardRootAddr* is found in figure 11. The assert makes sure that *forwardRootAddr* is in fact called with the next_root so that incrementing next_root correctly reflects the fact that r has been forwarded. The pre-condition for *forwardAddr* is satisfied, and furthermore the invariant guarantees that *forwardAddr* has been called with an unforwarded address. Executing *forwardAddr* implies that *isForwardStep* holds, and modifies *r, which means the old address is effectively removed from the roots and the new one added, so the post-condition holds. Incrementing next_root maintains the invariant involving which roots have been forwarded.

4.7. scanUnscanned

```
void scanUnscanned(void)
{
  addr n;

  while ((n = nextUnscannedNode()) != aNIL) {
    scanAddr(n);
  }
}
```

Showing that *scanUnscanned* satisfies its specification (figure 12) requires showing both partial correctness and loop termination, assuming that the pre-condition for *scanUnscanned* holds. The loop condition (LC), and loop invariant (LI), and a monotonically increasing metric (M) are:

$LC == mem'.unscanned! = \{\}$

$LI == mem\hat{}.roots = mem'.roots \wedge memEquiv(mem\hat{}, mem') \wedge addrsEquiv(reachable(mem\hat{}, mem\hat{}.roots),$
$reachable(mem', copiedNodes(mem')), mem\hat{}, mem')$

$M == size(nodesScanned(mem))[<= size(allNodes(mem))]$

Before the loop executes LI holds since

$mem' = mem\hat{} \Rightarrow$
$mem\hat{}.roots = mem'.roots \wedge memEquiv(mem\hat{}, mem')$
$\{\} = mem\hat{}.scanned \Rightarrow copiedNodes(mem') = unscanned$

$mem\hat{}.roots = mem\hat{}.unscanned$
$\Rightarrow addrsEquiv(reachable(mem\hat{}, mem\hat{}.roots),$
$reachable(mem', copiedNodes(mem')), mem\hat{}, mem')$

LI satisfies the pre-condition for *nextUnscannedNode*. The post-condition of *nextUnscannedNode* along with LI satisfies the pre-condition for *scanAddr*. The post-condition of *scanAddr* implies LI since *isScanStep* implies that the roots stay constant and that the memories are equivalent and the reachability condition is an explicit part of the post-condition of *scanAddr*.

If the loop terminates then ¬LC ∧ LI hold and the following parts of the post-condition for *scanUnscanned* can easily be discharged:

$mem\hat{}.roots = mem'.roots \Rightarrow mem\hat{}.roots = mem'.roots$
$mem'.unscanned! = \{\} \Rightarrow mem'.unscanned = \{\}$
$memEquiv(mem\hat{}, mem') \Rightarrow memEquiv(mem\hat{}, mem')$

I can simplify the remaining conjunct of the post-condition by noting: $mem'.unscanned = \{\}$
$\wedge\ addrsEquiv(reachable(mem\hat{}, mem\hat{}.roots),$
$reachable(mem', copiedNodes(mem')), mem\hat{}, mem') \Rightarrow$
$addrsEquiv(reachable(mem\hat{}, mem\hat{}.roots),$
$reachable(mem', mem'.scanned), mem\hat{}, mem')$

Leaving us to show that
$LC \wedge LI \Rightarrow reachable(mem', mem'.scanned) = mem'.scanned$

Each element in mem'.scanned satisfies *isScannedAddr* which means all of its pointers satisfy *isForwardedAddr* and thus are either in mem'.scanned or mem'.unscanned. But mem'.unscanned is empty, so every address referenced by an address in mem'.scanned must in mem'.scanned as well. This means reachable(mem', mem'.scanned) = mem'.scanned.

The loop terminates, because each execution of *scanAddr* adds a node to the scanned set, and the number of nodes which can be added to the scanned set is bounded by the total number of nodes.

4.8. nextUnscannedNode

```
addr nextUnscannedNode() {
  if (mem.scanned >= mem.unscanned) return aNIL;
  return mem.scanned;
}
```

The specification of *nextUnscannedNode* is found in figure 13. As captured in the abstraction function unscanned = [scanned..unscanned). Thus if mem.scanned >= mem.unscanned then {} = unscanned, and aNil should be returned. Otherwise an element of unscanned, mem.scanned, is returned as required by the post-condition. The specification could be satisfied by returning any unscanned element, but this implementation manages the unscanned set as a queue, with *nextUnscannedNode* returning the head of the queue.

4.9. scanAddr

```
void scanAddr(addr n){
  assert(n == mem.scanned);
  forwardAddr(&mem.to[n].car);
  forwardAddr(&mem.to[n].cdr);
  mem.scanned++;
}
```

The specification of *scanAddr* is found in figure 14. The assert makes sure that n is the location of the first element of the unscanned set and thus that incrementing scanned moves the node located at n from unscanned to scanned. The pre-condition for each *forwardAddr* is satisfied and the invariant guarantees that each is called with an unforwarded address, since all nodes at addresses at or above mem.scanned are guaranteed to contain only unforwarded addresses. The post-condition for *forwardAddr* implies that both the car and the cdr are forwarded and that *memEquiv* holds. Since *forwardAddr* only modifies the address passed to it, the roots are unchanged. Incrementing scanned moves n into the scanned set without changing the rest of memory. Taken together the last three points mean that *isScanStep* holds. The reachability condition is satisfied because n is in the copied set and the two *forwardAddrs* at most add the car and cdr to the copied set so the nodes reachable from the copied set are not changed. The invariant is maintained because the addresses in n are now forwarded, and scanned greater than n, indicating that n is in the scanned set.

4.10. forwardAddr

```
void forwardAddr(addr *a){
  if (mem.from[*a].tag != FWD) copyAddr(*a);
  *a = mem.from[*a].fwd;
}
```

The specification of *forwardAddr* is found in figure 16. In this implementation *forwardAddr* is called only with unforwarded nodes since in both places *forwardAddr* is used, the invariant states that the addresses are unforwarded. The specified interface is more general to allow the specification to be more broadly applicable. Given that a is unforwarded, *forwardAddr* must ensure *isForwardStep*. If the node has not been copied, then its tag is CONS and the pre-condition for *copyAddr* holds. This along with the post-condition of *copyAddr* implies that the $(addrUncopied(a, m) \Rightarrow isCopyStep(a, m, m'))$ $\wedge\, isCopiedAddr(a, a', m, m')$ conjuncts of *isForwardStep* hold. If the tag is FWD then $isCopiedAddr(a, a', m, m')$ already holds. The pointer update makes m'.forwarded[a] = a' hold. Neither of these things changes the equivalence between memories. They also do not change the scanned set, so *isForwardStep* holds and *forwardAddr* satisfies its specification.

4.11. copyAddr

```
void copyAddr(addr a) {
  mem.to[mem.unscanned] = mem.from[a];
  mem.from[a].tag = FWD;
  mem.from[a].fwd = mem.unscanned;
  mem.unscanned++;
}
```

The specification of *copyAddr* is found in figure 18. *addrFree* is satisfied because the node is copied to mem.unscanned which points to a free location, *addrUnforwarded* is satisfied because mem.unscanned is not forwarded. The roots and scanned sets are unchanged. Setting mem.from[a].tag = FWD removes the node from uncopied. Incrementing unscanned adds the new address to unscanned. Copying the node to unscanned rebinds it in memory. Finally, mem.from[a].fwd = mem.unscanned adds the new address to the forwarding map. None of this changes the equivalence of the memories. Choosing unscanned as the location to copy the node to completes the breadth-first management of the unscanned set, with unscanned acting as the tail of the queue. Since the node is copied to a location at or above scanned, and it contains only unforwarded addresses,. the invariant holds.

5. Application to Other Garbage Collectors

The implementation in Section 4 shows in detail how the specification applies to a simple two-space copying collector. It also applies to other copying based collectors including generational copying collectors, incremental copying collectors, and collectors which do not use breadth-first traversal of the node graph. This section considers how the specification applies to these variations of CGC.

5.1. Generational

Generational collectors attempt to minimize the cost of garbage collection by concentrating their efforts on those portions of memory that are most likely to contain garbage. Typically these are the portions of memory that have been most recently allocated. Generational collectors divide data into a number of generations that group the data by how old it is. They then collect younger generations more frequently than the older ones. [11]

The specification above can be used to describe generational collectors by simply choosing what to consider as the roots. For simple collectors, the roots are the global data structures, the stack, and the registers. For a generational collector the roots must also include any pointers from other generations into the one being collected. Tracking these inter-generational pointers is one of the major design issues in implementing a generational collector, but lies outside the scope of this specification. Given a set of roots that includes all the needed inter-generational pointers, the specification can stand without change.

5.2. Non-breadth first

Some research has been done on collectors that do not use a breadth-first traversal of the node graph [12] [9]. The intention is to improve locality by clustering closely connected portions of the node graph. Since data are accessed by following pointers, a copying strategy that copies subtrees of the graph so that they are physical close to each other may have this effect.

These collectors can still be described with the specification above. Two techniques can be used to change the order in which nodes are copied. One technique is to make the implementation's representation of the scanned set more complicated so as to allow nodes above the unscanned pointer to be in the scanned set. Since the specification does not dictate the representation of the scanned or unscanned set it is applicable to this technique. The other technique is to keep the scanned and unscanned sets representation as is, but to allow references in nodes in the unscanned set to be forwarded. This eager forwarding can change the order of copying without otherwise changing the basic algorithm, as long as *forwardAddr* can ignore already forwarded references. This is why *forwardAddr* is specified so that it can ignore already forwarded references.

5.3. Incremental

Incremental collectors work by interleaving collection with mutation. Recently work has been done on a new incremental copying collector that has some unusual properties with respect to the handling of roots [10]. When an incremental collection starts, it uses the roots as "hints" about what to copying, but does not forward them since that would violate certain invariants needed by the mutator. As the collection proceeds, the collector periodically resamples the roots for new parts of the graph to be copied. When the incremental collection is completed, the roots are forwarded, and the spaces "flipped".

The modifications to the specification to accommodate this incremental collector would be more extensive. The roots would have to consist of the union of all the roots sampled during collection. An interface to allow the copying of a subset of the roots would be needed. The *forwardRoots* interface would need to change so that it could forward only a subset of the roots. The overall structure would need to change to allow for repeatedly copying roots and the scanning the unscanned portions, forwarding roots only at termination. In addition, many of the proofs of correctness would need to change.

I mention this style of collection because while working on the specification, I realized that the original incremental collector implementation was flawed in an important way. All the roots were sampled each time the incremental collector gets control. In fact for correctness, one only needs to guarantee that the transitive closure of the roots at a flip is copied. This is obvious from the specification of *fullGC*. One still must sample some of the roots to get the collector started, but once it is started, one only needs to resample when trying to finish. The sampling of unneeded roots may well lead to data being copied that does not need to be. This flaw has been corrected.

6. Related work

Considering its importance, there are surprisingly few published attempts at formalizing garbage collection. Even *The Definition of Standard ML* [8] a formal semantics of SML contains the statement

> There are no rules concerning disposal of inaccessible addresses ("garbage collection").

The notable exception to this lack is the work by Demmers et. al. [2]. Their work differs from mine in several important ways. First, they are concerned with characterizing what data is preserved by a garbage collector (notably conservative and/or generational collectors), rather than capturing the details of a particular algorithm. In fact, their framework should apply equally to CGC and MSGC, although in their paper, they apply it primarily to MSGC. In their terminology, my specification models a *precise* garbage collector, that is, one which retains exactly those nodes reachable from the roots. They are concerned with describing *imprecise* collectors, that is, ones which may retain some nodes which are not reachable from the roots. They show that such imprecise collectors can be described by a precise collection with an augmentation to the points-to relation. This is the same sense in which my specification models generational GC, by augmenting the roots with the needed inter-generational pointers. Another way that their work differs from mine is in presentation, my formalization is presented in terms of a formal specification language, while their presentation uses more conventional mathematical notation. Finally they use their formalization to describe several implementations at a relatively high level of detail, while mine is used to prove the detailed correctness of a single simple collector.

Acknowledgments

I would like to thank Jim Horning and Jeannette Wing for teaching me Larch and in general for help with the specification. Jeannette Wing provided invaluable editorial assistence as well. Jim Horning served as my supervisor during a summer internship at DEC SRC, where this work was begun. I would like to thank him and DEC SRC for giving me the opportunity to spend the summer at SRC.

References

[1] C. J. Cheney.
A nonrecursive list compaction algorithm.
Communications of the ACM, 13(11):677–78, November 1970.

[2] A. Demers, M. Weiser, B. Hayes, H. Boehm, D. Bobrow, , and S. Shenker.
Combining generational and conservative garbage collection: Framework and implementations.
In *17th Annual ACM Symp. on Principles of Programming Languages*, pages 261–269, January 1990.

[3] Robert R. Fenichel and Jerome C. Yochelson.
A LISP garbage collector for virtual-memory computer systems.
Communications of the ACM, 12(11):611–612, November 1969.

[4] John V. Guttag and James J. Horning.
A tutorial on Larch and LCL, a Larch/C interface language.
In S. Prehn and W. J. Toetenel, editors, *VDM91: Formal Software Development Methods*, 10 1991.

[5] J.V. Guttag, J.J. Horning, and Andrés Modet.
Report on the Larch Shared Language: Version 2.3.
Report 58, DEC Systems Research Center, Palo Alto, CA, April 14, 1990.

[6] J.V. Guttag, J.J. Horning, and J.M. Wing.
Larch in five easy pieces.
TR 5, DEC SRC, 7 1985.

[7] John McCarthy.
Recursive functions of symbolic expressions and their computation by machine.
Communications of the ACM, 3(4):184–195, April 1960.

[8] Robin Milner, Mads Tofte, and Robert Harper.
The Definition of Standard ML.
MIT Press, 1989.

[9] David Moon.
Garbage collection in large lisp systems.
In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 235–246, August 1984.

[10] Scott Nettles, James O'Toole, David Pierce, and Nicholas Haines.
Replication-based incremental copying collection.
In *International Workshop on Memory Managment*. Springer-Verlag, September 1992.
Springer-Verlag Lecture Notes in Computer Science. To appear.

[11] Paul R. Wilson.
Uniprocessor garbage collection techniques.
In *International Workshop on Memory Managment*. Springer-Verlag, September 1992.
Springer-Verlag Lecture Notes in Computer Science. To appear.

[12] Paul R. Wilson, Micheal S. Lam, and Thomas G. Moher.
Effective static-graph reorganization to improve locality in garbage-collected systems.
In *Proceedings of the SIGPLAN Symposium on Programming Language Design and Implementation*, pages 177–191, June 1991.

7. Appendix

7.1. Traits

Address : **trait**
  includes *Set*(*A*, *SA*) % Sets of Addresses

Node : **trait**
  includes *Address*
  includes *Set*(*Val*, *SV*) % Sets of Values
  includes *Set*(*N*, *SN*) % Sets of Nodes
  N **tuple of** *id* : *UID*, *addrs* : *SA*, *vals* : *SV*

MemoryMain : **trait**
  includes *Node*
  includes *FiniteMappingAux*(*ANMap*, *A*, *N*, *SA* **for** *SDomain*)
  includes *FiniteMappingAux*(*AAMap*, *A*, *A*, *SA* **for** *SRange*, *SA* **for** *SDomain*)
  includes *TestSet1Arg*(*isValidAddr*, *isValidAddrSet*, *A*, *SA*, *M*)
  M **tuple of** *roots* : *SA*,
    uncopied : *SA*,
    unscanned : *SA*,
    scanned : *SA*,
    mem : *ANMap*,
    forwarded : *AAMap*
  introduces
    $isValidMemory : M \rightarrow Bool$
    $isValidAddr : A, M \rightarrow Bool$
    $effectiveAddr : A, M \rightarrow A$
  asserts
    $\forall\ m : M, a : A, n : N$
      $isValidMemory(m) ==$
        $isOneToOne(m.mem) \land isOneToOne(m.forwarded)$
        $\land isValidAddrSet(m.roots, m)$
        $\land m.uncopied \cap m.unscanned = \{\}$
        $\land m.uncopied \cap m.scanned = \{\}$
        $\land m.unscanned \cap m.scanned = \{\}$
        $\land m.uncopied \cap domain(m.forwarded) = \{\}$
        $\land m.unscanned \cup m.scanned = range(m.forwarded)$
        $\land m.uncopied \cup m.unscanned \cup m.scanned$
        $= domain(m.mem)$

      $isValidAddr(a, m) ==$ **if** $defined(m.forwarded, a)$
        then $defined(m.mem, m.forwarded[a])$
        else $defined(m.mem, a)$

      $effectiveAddr(a, m) ==$ **if** $defined(m.forwarded, a)$
        then $m.forwarded[a]$
        else a
  implies
    converts *isValidMemory*, *isValidAddr*, *isValidAddrSet*, *effectiveAddr*

Equiv : **trait**
  includes *Memory*
  includes *PairwiseElementTest2Arg*(*isEquivAddr*, *A*, *A*, *SA*, *SA*, *M*, *M*,
    addrsEquiv **for** *allPass*)
  introduces
    $isEquivAddr : A, A, M, M \rightarrow Bool$
    $isEquivNode : N, N, M, M \rightarrow Bool$
    $memEquiv : M, M \rightarrow Bool$
  asserts
    $\forall\ m, m' : M, a, a' : A, n, n' : N$
      $isEquivAddr(a, a', m, m') ==$
        $effectiveAddr(a, m') = effectiveAddr(a', m')$
        $\wedge isEquivNode(nodeAtAddr(a, m), nodeAtAddr(a', m'), m, m')$

      $isEquivNode(n, n', m, m') ==$
        $n.id = n'.id$
        $\wedge n.vals = n'.vals$
        $\wedge addrsEquiv(n.addrs, n'.addrs, m, m')$

      $memEquiv(m, m') ==$
        $isValidMemory(m) \wedge isValidMemory(m')$
        $\wedge addrsEquiv(m.roots, m'.roots, m, m')$
        $\wedge addrsEquiv(allNodes(m), allNodes(m'), m, m')$
  implies
    converts *isEquivAddr*, *isEquivNode*, *addrsEquiv*, *memEquiv*

Reachable : **trait**
  includes *Memory*
  introduces
    $reachable : SA, M \rightarrow SA$
    $r_1 : SA, SA, M \rightarrow SA$
  asserts
    $\forall\ m : M, a : A, as, as_1, as_2 : SA$
      $reachable(as, m) == r_1(\{\}, as, m)$

      $r_1(as, \{\}, m) == as$
      $r_1(as_1, insert(a, as_2), m) ==$
        $r_1(insert(a, as_1),$
        $(as_2 \cup (m.mem[a]).addrs) - insert(a, as_1), m)$
  implies
    converts *reachable*, r_1

GC : **trait**
  includes *Memory*
  includes *Equiv*
  includes *Reachable*
  introduces
    isFullGC : $M, M \rightarrow Bool$
    isInitialMemory : $M \rightarrow Bool$
    isFinalGCMemory : $M \rightarrow Bool$
  asserts
    $\forall\ m, m' : M$
      $isFullGC(m, m') ==$
        $isInitialMemory(m)$
        $\wedge isFinalGCMemory(m')$
        $\wedge memEquiv(m, m')$
        $\wedge addrsEquiv(reachable(m.roots, m), m'.scanned, m, m')$

      $isInitialMemory(m) ==$
        $isValidMemory(m)$
        $\wedge \{\} = m.unscanned$
        $\wedge \{\} = m.scanned$
        $\wedge \{\} = m.forwarded$

      $isFinalGCMemory(m) ==$
        $isValidMemory(m)$
        $\wedge \{\} = m.unscanned$
        $\wedge \{\} = rootsUnforwarded(m)$
  implies
    converts *isFullGC*, *isInitialMemory*, *isFinalGCMemory*

Scan : **trait**
  includes *Memory*
  includes *Forward*
  introduces
    isScannedAddr : $A, M \rightarrow Bool$
    isScanStep : $A, M, M \rightarrow Bool$
  asserts
    $\forall\ m, m' : M, a : A$
      $isScannedAddr(a, m) ==$
        $addrScanned(a, m)$
        $\wedge isForwardedAddr(a, a, m, m)$
        $\wedge isForwardedSet((m.mem[a]).addrs,$
        $(m.mem[a]).addrs, m, m)$

      $isScanStep(a, m, m') ==$
        $memEquiv(m, m')$
        $\wedge m.roots = m'.roots$
        $\wedge addrUnscanned(a, m)$
        $\wedge addrScanned(a, m')$
        $\wedge isForwardedSet((m.mem[a]).addrs,$
        $(m'.mem[a]).addrs, m, m')$
  implies
    $\forall\ m, m' : M, a : A$

isScanStep$(a, m, m') \Rightarrow$ *isScannedAddr*(a, m')

converts *isScannedAddr*, *isScanStep*

Forward : **trait**
includes *Memory*
includes *Copy*
includes *PairwiseElementTest2Arg*(*isForwardedAddr*, A, A, SA, SA, M, M,
isForwardedSet **for** *allPass*)
introduces
isForwardedAddr : $A, A, M, M \rightarrow$ *Bool*
isForwardStep : $A, A, M, M \rightarrow$ *Bool*
asserts
$\forall\ m, m' : M, a, a' : A$
isForwardedAddr(a, a', m, m') ==
isCopiedAddr(a, a', m, m')
$\wedge$*effectiveAddr*$(a, m') = a'$

isForwardStep(a, a', m, m') ==
memEquiv(m, m')
$\wedge$*addrUnforwarded*(a, m)
$\wedge$*addrForwarded*(a', m')
$\wedge$(*addrUncopied*$(a, m) \Rightarrow$ *isCopyStep*(a, m, m'))
$\wedge$*isCopiedAddr*(a, a', m, m')
$\wedge m'$.*forwarded*$[a] = a'$
$\wedge m$.*scanned* $= m'$.*scanned*
implies
$\forall\ m, m' : M, a, a' : A$
isForwardStep$(a, a', m, m') \Rightarrow$ *isForwardedAddr*(a, a', m, m')

converts *isForwardedAddr*, *isForwardStep*

Copy : **trait**
includes *Memory*
includes *Equiv*
introduces
isCopiedAddr : $A, A, M, M \rightarrow$ *Bool*
isCopyStep : $A, M, M \rightarrow$ *Bool*
asserts
$\forall\ m, m' : M, a, a' : A$
isCopiedAddr(a, a', m, m') ==
isEquivAddr(a, a', m, m')
$\wedge$*addrCopied*(*effectiveAddr*$(a', m'), m')$

isCopyStep(a, m, m') ==
memEquiv(m, m')
$\wedge$*addrUncopied*(a, m)
$\wedge$*addrFree*$((m'$.*forwarded*$[a]), m)$
$\wedge$*addrUnforwarded*$(m'$.*forwarded*$[a], m)$
$\wedge m' = [m$.*roots*,

$delete(a, m.uncopied),$
$insert(m'.forwarded[a], m.unscanned),$
$m.scanned,$
$rebind(m.mem, a, m'.forwarded[a]),$
$bind(m.forwarded, a, m'.forwarded[a]))]$

implies

$\forall\ m, m' : M, a : A$

$isCopyStep(a, m, m') \Rightarrow isCopiedAddr(a, a, m, m')$

converts *isCopiedAddr*, *isCopyStep*

These are some of the less important traits which were not discussed in the text.

Memory : **trait**
includes *MemoryMain*
includes *MemoryAuxiliary*

MemoryAuxiliary : **trait**
includes *MemoryMain*
includes *SetOps*
includes *ElementTest*(*addrUnforwarded*, *A*, *SA*, *M*, *addrsUnforwarded* **for** *filter*)
introduces

$nodeAtAddr : A, M \rightarrow N$
$allNodes : M \rightarrow SA$
$copiedNodes : M \rightarrow SA$
$addrFree : A, M \rightarrow Bool$
$addrCopied : A, M \rightarrow Bool$
$addrUncopied : A, M \rightarrow Bool$
$addrUnscanned : A, M \rightarrow Bool$
$addrScanned : A, M \rightarrow Bool$
$addrForwarded : A, M \rightarrow Bool$
$addrUnforwarded : A, M \rightarrow Bool$
$addrRoot : A, M \rightarrow Bool$
$rootsUnforwarded : M \rightarrow SA$

asserts

$\forall\ m : M, a : A, n : N$

$nodeAtAddr(a, m) == m.mem[effectiveAddr(a, m)]$

$allNodes(m) == m.uncopied \cup m.unscanned \cup m.scanned$

$copiedNodes(m) == m.unscanned \cup m.scanned$

$addrFree(a, m) == \neg defined(m.mem, a)$

$addrCopied(a, m) == a \in copiedNodes(m)$
$\vee defined(m.forwarded, a)$

$addrUncopied(a, m) == a \in m.uncopied$

$addrUnscanned(a, m) == a \in m.unscanned$

$addrScanned(a, m) == a \in m.scanned$

$addrForwarded(a, m) == a \in copiedNodes(m)$

$addrUnforwarded(a, m) == a \in m.uncopied$
$\quad \vee defined(m.forwarded, a)$

$addrRoot(a, m) == a \in m.roots$

$rootsUnforwarded(m) == addrsUnforwarded(m.roots, m)$

implies
converts $nodeAtAddr$, $allNodes$, $copiedNodes$, $addrFree$, $addrCopied$, $addrUncopied$, $addrUnscanned$, $addrScanned$, $addrForwarded$, $addrUnforwarded$, $addrRoot$, $rootsUnforwarded$, $addrsUnforwarded$

$FiniteMappingAux(Map, Domain, Range)$: **trait**
includes $FiniteMap(Map, Domain, Range)$
includes $Set(Domain, SDomain)$
includes $Set(Range, SRange)$
introduces
$\_[\_] : Map, Domain \rightarrow Range$
$unbind : Map, Domain \rightarrow Map$
$rebind : Map, Domain, Domain \rightarrow Map$
$isOneToOne : Map \rightarrow Bool$
$bound : Map, Range \rightarrow Bool$
$domain : Map \rightarrow SDomain$
$range : Map \rightarrow SRange$
asserts
$\forall\ m : Map, d, d_1, d_2 : Domain, r, r_1, r_2 : Range$
$m[d] = apply(m, d)$

$unbind(\{\}, d_1) == \{\}$
$unbind(bind(m, d_1, r), d_2) ==$
$\quad$ **if** $d_1 = d_2$ **then** m
$\quad$ **else** $bind(unbind(m, d_2), d_1, r)$

$rebind(\{\}, d_1, d_2) = \{\}$
$rebind(bind(m, d, r), d_1, d_2) ==$
$\quad$ **if** $d = d_1$ **then** $bind(m, d_2, r)$
$\quad$ **else** $bind(rebind(m, d_1, d_2), d, r)$

$isOneToOne(\{\})$
$isOneToOne(bind(m, d, r)) == \neg bound(m, r) \wedge isOneToOne(m)$

$\neg bound(\{\}, r)$
$bound(bind(m, d, r_1), r_2) == r_1 = r_2 \vee bound(m, r_2)$

$domain(\{\}) == \{\}$
$domain(bind(m, d, r)) == insert(d, domain(m))$

$range(\{\}) == \{\}$
$range(bind(m, d, r)) == insert(r, range(m))$

implies
$\forall\ m : Map, d_1 : Domain$
$\neg defined(unbind(m, d_1), d_1)$

converts $unbind, rebind, isOneToOne, bound, \_[\_], domain, range$

$PairwiseElementTest2Arg(pass, E_1, E_2, S_1, S_2, T_1, T_2)$: **trait**
assumes $Set(E_1, S_1)$
assumes $Set(E_2, S_2)$
introduces
$pass : E_1, E_2, T_1, T_2 \rightarrow Bool$
$allPass : S_1, S_2, T_1, T_2 \rightarrow Bool$
$onePasses : E_1, S_2, T_1, T_2 \rightarrow Bool$
$removePassing : E_1, S_2, T_1, T_2 \rightarrow S_2$
asserts
$\forall\ s_1 : S_1, s_2 : S_2, e_1 : E_1, e_2 : E_2, t_1 : T_1, t_2 : T_2$
$allPass(\{\}, \{\}, t_1, t_2)$
$allPass(insert(e_1, s_1), s_2, t_1, t_2) ==$
$onePasses(e_1, s_2, t_1, t_2)$
$\wedge allPass(s_1, removePassing(e_1, s_2, t_1, t_2), t_1, t_2)$

$\neg onePasses(e_1, \{\}, t_1, t_2)$
$onePasses(e_1, insert(e_2, s_2), t_1, t_2) ==$
$pass(e_1, e_2, t_1, t_2)$
$\vee onePasses(e_1, s_2, t_1, t_2)$

$removePassing(e_1, \{\}, t_1, t_2) == \{\}$
$removePassing(e_1, insert(e_2, s_2), t_1, t_2) ==$
if $pass(e_1, e_2, t_1, t_2)$
then s_2
else $insert(e_2, removePassing(e_1, s_2, t_1, t_2))$
implies
converts $allPass, onePasses, removePassing$

$TestSet1Arg(elemOp, setOp, E, SE, A)$: **trait**
assumes $Set(E, SE)$
introduces
$setOp : SE, A \rightarrow Bool$
$elemOp : E, A \rightarrow Bool$
asserts $\forall\ e : E, se : SE, a : A$
$setOp(\{\}, a)$
$setOp(insert(e, se), a) == setOp(se, a) \wedge elemOp(e, a)$
implies converts $setOp$

These are traits from the LSL handbook[1].

Set(E, C) : **trait**

includes

SetBasics,

Natural(N),

DerivedOrders$(C, \subseteq$ **for** $\leq, \supseteq$ **for** $\geq, \subset$ **for** $<, \supset$ **for** $>)$

introduces

$delete : E, C \rightarrow C$

$\{\_\} : E \rightarrow C$

$\_ \cup \_, \_ \cap \_, \_ - \_ : C, C \rightarrow C$

$size : C \rightarrow N$

asserts

$\forall\ e, e_1, e_2 : E, s, s_1, s_2 : C$

$\{e\} == insert(e, \{\})$

$e_1 \in delete(e_2, s) == e_1 \neq e_2 \wedge e_1 \in s$

$e \in (s_1 \cup s_2) == e \in s_1 \vee e \in s_2$

$e \in (s_1 \cap s_2) == e \in s_1 \wedge e \in s_2$

$e \in (s_1 - s_2) == e \in s_1 \wedge e \notin s_2$

$size(\{\}) == 0$

$size(insert(e, s)) ==$ **if** $e \notin s$ **then** $size(s) + 1$ **else** $size(s)$

$s_1 \subseteq s_2 == s_1 - s_2 = \{\}$

implies

AbelianMonoid$(\cup$ **for** $\circ, \{\}$ **for** $unit, C$ **for** $T)$,

$AC(\cap, C)$,

JoinOp$(\cup)$,

MemberOp,

PartialOrder$(C, \subseteq$ **for** $\leq, \supseteq$ **for** $\geq, \subset$ **for** $<, \supset$ **for** $>)$,

UnorderedContainer

C **generated by** $\{\}, \{\_\}, \cup$

$\forall\ e : E, s, s_1, s_2 : C$

$insert(e, s) \neq \{\}$

$insert(e, insert(e, s)) == insert(e, s)$

$s_1 \subseteq s_2 == s_1 - s_2 = \{\}$

converts $\in, \notin, \{\_\}, delete, size, \cup, \cap, -, \subseteq, \supseteq, \subset, \supset$

SetBasics(E, C) : **trait**

introduces

$\{\} : \rightarrow C$

$insert : E, C \rightarrow C$

$\_ \in \_, \_ \notin \_ : E, C \rightarrow Bool$

asserts

C **generated by** $\{\}, insert$

C **partitioned by** $\in$

$\forall\ s : C, e, e_1, e_2 : E$

$e \notin s == \neg(e \in s)$

$e \notin \{\}$

$e_1 \in insert(e_2, s) == e_1 = e_2 \vee e_1 \in s$

implies

UnorderedContainer,

MemberOp

$\forall\, e, e_1, e_2 : E, s : C$

$\quad insert(e, s) \neq \{\}$

$\quad insert(e, insert(e, s)) == insert(e, s)$

converts $\in, \notin$

SetOps : **trait**

assumes

Countable,

SetBasics

includes *CollectionOps*(*false* **for** *dups*)

introduces

$delete : E, C \rightarrow C$

$\_\cup\_, \_\cap\_ : C, C \rightarrow C$

asserts

$\forall\, e, e_1, e_2 : E, s, s_1, s_2 : C$

$\quad e_1 \in delete(e_2, s) == e_1 \neq e_2 \wedge e_1 \in s$

$\quad e \in (s_1 \cup s_2) == e \in s_1 \vee e \in s_2$

$\quad e \in (s_1 \cap s_2) == e \in s_1 \wedge e \in s_2$

$\quad e \in (s_1 - s_2) == e \in s_1 \wedge e \notin s_2$

implies

AbelianMonoid($\cup$ **for** $\circ$, $\{\}$ **for** *unit*, C **for** T),

AC($\cap$, C),

JoinOp($\cup$),

PartialOrder(C, $\subseteq$ **for** $\leq$, $\supseteq$ **for** $\geq$, $\subset$ **for** $<$, $\supset$ **for** $>$)

C **generated by** $\{\}, \{\_\}, \cup$

$\forall\, e : E, s, s_1, s_2 : C$

$\quad size(insert(e, s)) ==$ **if** $e \in s$ **then** $size(s)$ **else** $succ(size(s))$

$\quad s_1 \subseteq s_2 == s_1 - s_2 = \{\}$

converts $\in, \notin, \{\_\}, delete, size, \cup, \cap, -, \subseteq, \supseteq, \subset, \supset$

ElementTest(*pass*, E, C, T) : **trait**

assumes *Container*

introduces

$pass : E, T \rightarrow Bool$

$somePass : C, T \rightarrow Bool$

$allPass : C, T \rightarrow Bool$

$filter : C, T \rightarrow C$

asserts $\forall\, c : C, e : E, t : T$

$\quad \neg somePass(\{\}, t)$

$\quad somePass(insert(e, c), t) == pass(e, t) \vee somePass(c, t)$

$\quad allPass(\{\}, t)$

$\quad allPass(insert(e, c), t) == pass(e, t) \wedge allPass(c, t)$

$\quad filter(\{\}, t) == \{\}$

$\quad filter(insert(e, c), t) ==$

$\qquad$ **if** $pass(e, t)$ **then** $insert(e, filter(c, t))$ **else** $filter(c, t)$

implies converts *somePass*, *allPass*, *filter*

FiniteMap(M, D, R) : **trait**
introduces
$\{\} : \rightarrow M$
$bind : M, D, R \rightarrow M$
$apply : M, D \rightarrow R$
$defined : M, D \rightarrow Bool$
asserts
M **generated by** $\{\}$, *bind*
M **partitioned by** *apply*, *defined*
$\forall\ m : M, d, d_1, d_2 : D, r : R$
$apply(bind(m, d_2, r), d_1) == \textbf{if } d_1 = d_2 \textbf{ then } r \textbf{ else } apply(m, d_1)$
$\neg defined(\{\}, d)$
$defined(bind(m, d_2, r), d_1) == (d_1 = d_2) \vee defined(m, d_1)$
implies
converts *apply*, *defined*
exempting $\forall\ d : D apply(\{\}, d)$

7.2. Interfaces

```
imports base;
uses GC(memory for M, addr for A );

void gc(void) memory mem; {
    requires isInitialMemory(mem^);
    modifies mem;
    ensures
         isFullGC(mem^, mem')
      /\ isFinalGCMemory(mem');
}
```

```
imports base;
uses GC(memory for M, addr for A);

void finalizeGC(void) memory mem;  {
  requires isFinalGCMemory(mem^);
  modifies mem;
  ensures
       isInitialMemory(mem')
    /\ mem^.scanned = mem'.uncopied
    /\ mem^.roots = mem'.roots
    /\ mem^.mem = mem'.mem;
}
```

```
imports base;
uses GC(memory for M, addr for A);

void forwardRoots(void) memory mem;  {
  requires isInitialMemory(mem^);
  modifies mem;
```

```
  ensures
       {} = rootsUnforwarded(mem')
    /\ mem'.roots = mem'.unscanned
    /\ {} = mem'.scanned
    /\ memEquiv(mem^, mem');
}


imports base;

addr nextUnforwardedRoot(void) memory mem; {
  requires isValidMemory(mem^);
  ensures if {} = rootsUnforwarded(mem^)
          then result = aNIL
          else result \in rootsUnforwarded(mem^);
}


uses Forward(memory for M, addr for A );

void forwardRootAddr(addr *a) memory mem; {
  requires
       isValidMemory(mem^) /\ (*a)^ \in rootsUnforwarded(mem^);
  modifies mem, *a;
  ensures
       mem'.roots = (mem^.roots - {(*a)^}) \U {(*a)'}
    /\ isForwardStep((*a)^, (*a)', mem^, mem');
}


imports base;

void scanUnscanned(void) memory mem;  {
  requires
       {} = rootsUnforwarded(mem^)
    /\ mem^.roots = mem^.unscanned
    /\ {} = mem^.scanned
    /\ isValidMemory(mem^);
  modifies mem;
  ensures
       mem^.roots = mem'.roots
    /\ mem'.unscanned = {}
    /\ memEquiv(mem^, mem')
    /\ addrsEquiv(reachable(mem^.roots, mem^),
                  mem'.scanned, mem^, mem');
}


imports base;

addr nextUnscannedNode(void) memory mem;  {
  requires isValidMemory(mem^);
  ensures if {} = mem^.unscanned
          then result = aNIL
          else result \in mem^.unscanned;
}
```

```
imports base;
uses Scan(memory for M, addr for A);

void scanAddr(addr a) memory mem;  {
  requires
        isValidMemory(mem^)
     /\ addrUnscanned(a, mem^)
     /\ addrsEquiv(reachable(mem^.roots, mem^),
                   reachable(copiedNodes(mem^), mem^),
                   mem^, mem^);
  modifies mem;
  ensures
        isScanStep(a, mem^, mem')
     /\ addrsEquiv(reachable(mem^.roots, mem^),
                   reachable(copiedNodes(mem'), mem'),
                   mem^, mem');
}
```

```
imports base;
uses Forward(memory for M, addr for A);

void forwardAddr(addr *a)  memory mem; {
    requires isValidMemory(mem^);
    modifies mem, *a;
    ensures
        if isForwardedAddr((*a)^, (*a)', mem^, mem')
        then (*a)^ = (*a)' /\ mem^ = mem'
        else isForwardStep((*a)^, (*a)', mem^, mem');
}
```

```
imports base;
uses Copy(memory for M, addr for A );

void copyAddr(addr a)  memory mem;  {
  requires
        isValidMemory(mem^)
     /\ addrUncopied(a, mem^);
  modifies mem;
  ensures isCopyStep(a, mem^, mem');
}
```

This is base.lcl.

```
abstract type addr;
constant addr aNIL;
abstract type memory;
memory mem;
uses Memory(memory for M, addr for A );
uses Reachable(memory for M, addr for A );
uses Equiv(memory for M, addr for A );
```